Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 54 (2005) 99–124

Science of Computer Programming

www.elsevier.com/locate/scico

Implementing advanced spoken dialogue management in Java

Ian O'Neill[a,*], Philip Hanna[a], Xingkun Liu[a], Des Greer[a], Michael McTear[b]

[a] *School of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, UK*
[b] *School of Computing and Mathematics, University of Ulster, Jordanstown, BT37 0QB, UK*

Received 15 January 2004; received in revised form 4 May 2004; accepted 20 May 2004

Available online 13 August 2004

Abstract

In this article we describe how Java can be used to implement an object-based, cross-domain, mixed initiative spoken dialogue manager (DM). We describe how dialogue that crosses between several business domains can be modelled as an inheriting and collaborating suite of objects suitable for implementation in Java. We describe the main features of the Java implementation and how the Java dialogue manager can be interfaced via the Galaxy software hub, as used in the DARPA-sponsored Communicator projects in the United States, with the various off-the-shelf components that are needed in a complete end-to-end spoken dialogue system. We describe the interplay of the Java components in the course of typical dialogue turns and present an example of the sort of dialogue that the Java DM can support.

Keywords: Object-orientation; Java; Spoken dialogue systems

1. Introduction

The following paper describes key features of a suite of Java classes for cross-domain, mixed initiative spoken dialogue management. The Java dialogue manager (DM) operates within the DARPA Communicator architecture based on the Galaxy hub, a software router

* Corresponding author.
E-mail addresses: i.oneill@qub.ac.uk (I. O'Neill), p.hanna@qub.ac.uk (P. Hanna), xingkun.liu@qub.ac.uk (X. Liu), des.greer@qub.ac.uk (D. Greer), mf.mctear@ulster.ac.uk (M. McTear).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.05.006

developed by the Spoken Language Systems group at MIT [9], subsequently released as an open source package in collaboration with the MITRE Corporation, and now available on SourceForge [6]. An object-model and dialogue management strategies developed for a Prolog++ standalone DM [10,11] served as the starting point for development of a system written in Java. Whereas the Prolog++ prototype interacted with the user by means of text-based Prolog structures, the new Java DM interacts via the Galaxy hub with off-the-shelf software components that support spoken interaction between system and user. The current Java implementation is being developed as part of an 18-month EPSRC project, which began in January 2003. The aim of the project is to explore the use of object-oriented software engineering techniques in the development of advanced spoken dialogue systems. Although some currently available dialogue systems use object components in accordance with the latest software engineering orthodoxy [1], little published research addresses the question of how established techniques of object-oriented software engineering [2,3] can contribute to the dialogue management task. It is hoped that our Java-based OO approach to spoken dialogue management will provide a framework within which generic confirmation strategies and rules-of-thumb specific to particular business domains can be intuitively maintained and extended. The present implementation features slot-filling dialogues, where the types of information required to complete the individual stages of a transaction are known to the developers in advance. However, we are currently investigating approaches that would incorporate a greater degree of advice-giving and problem-solving into the system's generic dialogue behaviour.

2. The nature of advanced spoken dialogue management

Advanced spoken dialogue systems are typically used to help telephone callers complete a transaction in a well-defined business domain. For example, a user may make a train timetable enquiry or book accommodation. Advanced dialogue systems have the advantage that they allow mixed initiative interactions—that is, they permit the caller to express his/her intentions using whatever words seem most appropriate and to provide more information or less information than the system requested. The dialogue manager in a spoken language system determines the system's response to the user's utterances. The system may confirm what the user has said, check to see if the user's request can be fulfilled (e.g. is there a train on the requested day at the requested time?), or ask for more information. The system may perform a combination of confirming, validating and requesting in a single dialogue turn, in much the same manner as a human dialogue partner. In mixed initiative dialogues that deal with more than one domain (e.g. enquiries about accommodation and events, and supporting exchanges about payment and addresses), the system must be able to identify the topic of the dialogue—since in a mixed initiative exchange the user is free to provide whatever information he or she deems appropriate at the particular dialogue turn (e.g. the user may start to ask about events while booking accommodation). Having identified the (ongoing) topic of the dialogue, the system must then apply appropriate dialogue management expertise.

3. Spoken dialogue systems and object-orientation

The attraction of object-orientation is that it can be used to separate generic dialogue behaviour from domain-specific behaviour. In an OO implementation, routine functionality such as turn-taking and confirming what the user has said is made available through inheritance to more specialised dialogue components. Maintainability is therefore enhanced: generic behaviour need be implemented only once; new agents can inherit the generic dialogue behaviour and possibly refine the specialised behaviour of existing experts: e.g. TheatreExpert can be introduced as a subclass of EventExpert. The more specialised components encapsulate 'expert rules' for enquiry processing in a particular business domain (e.g. "if a location and a hotel have been requested, then check what class of hotel is required") or rules that represent know-how that can be used in a number of business domains (e.g. how to elicit credit card details). In order to enable mixed initiative interactions across domains, we model the system's behaviour as a collaboration between a cohort of 'agents', each an instance of a Java class. An agent is a specialist in a particular transactional area—e.g. booking accommodation or eliciting an address—and uses its own domain-specific expert rules to elicit information (e.g. information for making a hotel booking) that is stored in a specialised dialogue frame. In our system, each agent thus encapsulates a skillset for a substantial dialogue or subdialogue. Other developers have also adopted an agent-based approach to dialogue, though sometimes dialogue agents each perform very simple tasks rather than engage in extensive discourse: in [14], for example, simple generic error-handling agents, based on Java and XML, ask the user to repeat misunderstood input.

In the Queen's Communicator the dialogue product (the knowledge to be elicited) is a tree-like structure, the nodes of which are complete blocks of information (e.g. details of a theatre reservation, or a set of credit card details). Each of these nodes is implemented as a specialisation of a generic DialogFrame (TheatreDialogFrame and CreditCardDialogFrame are two examples of these specialised dialogue frames). The CMU Communicator team adopts a similar approach, though in their branching dialogue product the nodes are individual data items rather than complete frames [13]. The tree-like dialogue product emerges from the fact that a dialogue frame may include as part of its attributes links to frames of other types—for example a TheatreDialogFrame, may include a link to a PaymentDialogFrame. Thus for example the `createAttributes( )` method in the `TheatreDialogFrame( )` constructor includes the following definition and initialisation:

```
Attribute paymentDetails = new Attribute("PaymentDetails",
                                         ...
                                         Attribute.LINKEDFRAME);
paymentDetails.setValue(new LinkedFrameAttributeValue("Payment"));
addAttributeToFrame(paymentDetails );                                (1)
```

The "placeholder string" `paymentDetails` indicates little more than that a frame of theatre booking information should include as an attribute a frame of payment details. It will fall to the DomainSpotter (described in Section 5.6) to determine which Expert

implementation will be used in a given discourse segment to elicit from the user the information needed to populate the dialogue frame, and the chosen Expert will in turn be responsible for selecting its preferred dialogue frame implementation. However, when it comes to speculating about the significance of an out-of-context utterance by the user (a comment about a credit card near the start of an accommodation booking, for example), the DomainSpotter can already use the placeholder strings to help determine which Expert might be used subsequently to further explore what the user has said. Again, the intention is to make the implementation as flexible as possible: new specialist objects may be added to the object hierarchy, and the DomainSpotter decides, as the discourse unfolds, which of the available Expert implementations to use. This is a key feature of the DM in the Queen's Communicator: the discourse structure and the corresponding dialogue product evolve dynamically as agents are selected by a DomainSpotter, in the light of the user's utterances or as a consequence of the agents' own rules. It is this process, rather than an overarching dialogue plan or agenda, that drives the discourse forward, sometimes across domain boundaries.

4. The DARPA Communicator architecture

Java is particularly attractive for implementation. As a fully fledged and relatively intuitive OO language it provides the means of implementing the inheriting and collaborating agent components that we require in our cross-domain, mixed initiative dialogue manager. Moreover, on a pragmatic note, Java is a language with which we are familiar (it is our universities' main teaching language for software engineering), and it comes with the software resources and development environments necessary for a serious software engineering task. However, a dialogue manager is only one of several main modules that are required to implement an end-to-end dialogue system. As a whole, the spoken dialogue system must

- recognise the words that the user says,
- attempt to determine the intention behind the words,
- decide how to respond, potentially using information from a database,
- have the 'conceptual' response generated as a well-formed natural language phrase,
- and utter the phrase as synthesised or concatenated speech.

Thus the dialogue manager, written in the developers' OO language of choice, Java, requires a means of communicating with the other, probably third-party modules that it needs for support—and whose implementation details need not concern the DM developers. These components typically include: an automatic speech recogniser, a semantic parser, a database 'back-end', a natural language generator, and a text-to-speech engine.

The DARPA Communicator architecture was conceived for the purpose of facilitating interaction between the different dialogue system modules. DARPA Communicator systems use a 'hub-and-spoke' architecture: each module seeks services from and provides services to the other modules by communicating with them through a central software router, the *Galaxy* hub. Java, along with C and C++ is supported in the API (application program interface) to the Galaxy hub. Thus an object-based dialogue manager written in

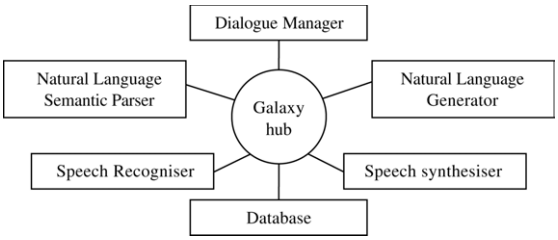

Fig. 1. DARPA Communicator architecture. *Separate 'server' interfaces between each component and the hub have been omitted.*

Java can interact via the Galaxy hub with the third-party modules that enable the system as a whole to hear and understand the user, to interact with a database and utter a response. The relationship between the dialogue manager, the Galaxy hub, and the other system modules is represented schematically in Fig. 1. Of immediate concern to a DM are the parser—which creates text-based semantic representations of key concepts in the user's utterance, and requires DM developers to create appropriate semantic grammars for it to work with—and the natural language generator (NLG)—which also requires rules to translate semantic representations emerging from the DM into natural language utterances. Organisations such as Carnegie Mellon University and the University of Colorado have already undertaken substantial development based on the Communicator architecture and made available complete working systems [5,7], including the hub-compliant components that support the DM. For the application developer wishing to create an original DM in Java, this off-the-shelf material is a valuable resource: existing components can simply be incorporated 'as is' into the overall spoken dialogue system, leaving the human dialogue modeller to concentrate on the development of core dialogue management functionality.

When it comes to interfacing the Java DM with the Galaxy hub, extensive documentation for the Java API is available [6]. In the Communicator architecture the main dialogue system modules act as both servers—service providers—and clients—service requesters, and the information they pass to each other is routed through the Galaxy hub in the form of 'hubframes'. To ensure that a hubframe requesting a particular service is routed to the appropriate server, the system developer creates a 'hubscript'. Where necessary, the hubscript also ensures that the interaction between the servers is appropriately sequenced.

Let us consider the manner in which the dialogue management module (an original suite of objects written in Java) of the Queen's Communicator system is integrated with the other system components (taken directly from the CU download [5]). The Java DialogueServer class, used by a DialogManager class to interface with the Galaxy hub (Fig. 2c), first of all includes the relevant hub package (*import galaxy.server.*;*) and then creates and populates frames of information in the Galaxy format. The hubscript associates 'keys' in the frame (*:nl_request* in the following example) with specific service providers (e.g. a *n*atural *l*anguage server) and routes the frame accordingly. In a method to send a phrase to a natural language server, the DialogueServer might include the lines:

```
// Create a new frame to send to the hub
GFrame newFrame = new Clause( "main" );
```

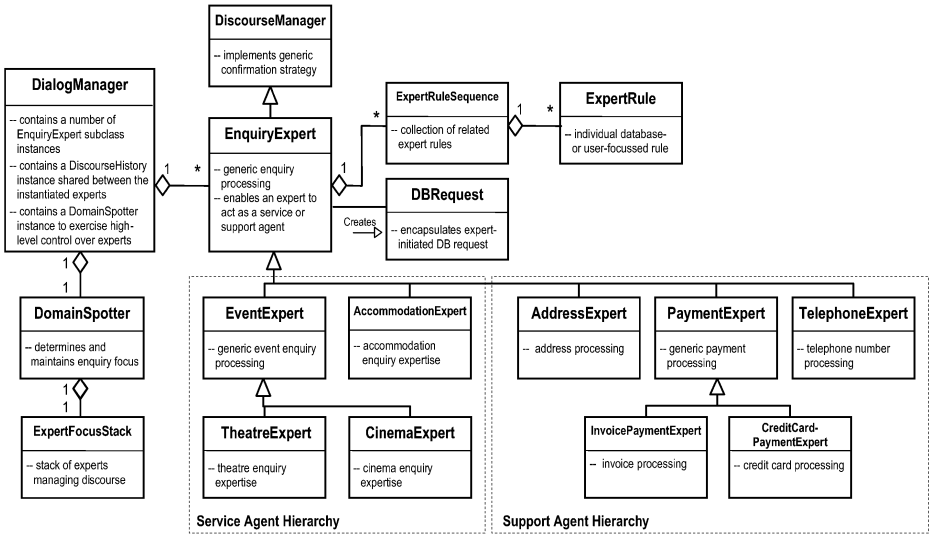

Fig. 2a. The Expertise Hierarchy.

```
newFrame.setProperty( ":nl_request", 1 );
newFrame.setProperty( ":prompt", message );
// Get the current environment and send the frame
try{ this.getCurrentEnvironment().writeFrame( newFrame );}
```
(2)

The hubscript in turn includes a line that causes the frame of data to be routed to the *process_nl_request* method of the natural language server.

The Queen's Communicator system currently accepts keyed natural language input. To semantically tag key words and phrases from the input we use the Phoenix natural language parser, originally developed at Carnegie Mellon University [15] and now incorporated into the CU Communicator download [5]. The semantically tagged words and phrases are then passed to the dialogue manager for processing. For output the dialogue manager uses the University of Edinburgh's Festival speech synthesiser [8] (again this is available as a component of the Colorado download) to 'speak' semantic concepts—e.g. "confirm_phrase: accommodation_type hotel". We are currently adding an in-house natural language generation module to produce naturalistic system utterances from the semantic concepts prior to synthesis. Likewise we are experimenting with an off-the-shelf speech recognition engine in order to generate input strings from the user's utterances.

5. The Java implementation

Our Java implementation of a dialogue manager leverages the strengths of object-orientation by separating generic behaviour from domain-specific behaviour [12]. Figs. 2a and 2b show the main components of the Java DM, and the main inheritance hierarchies.

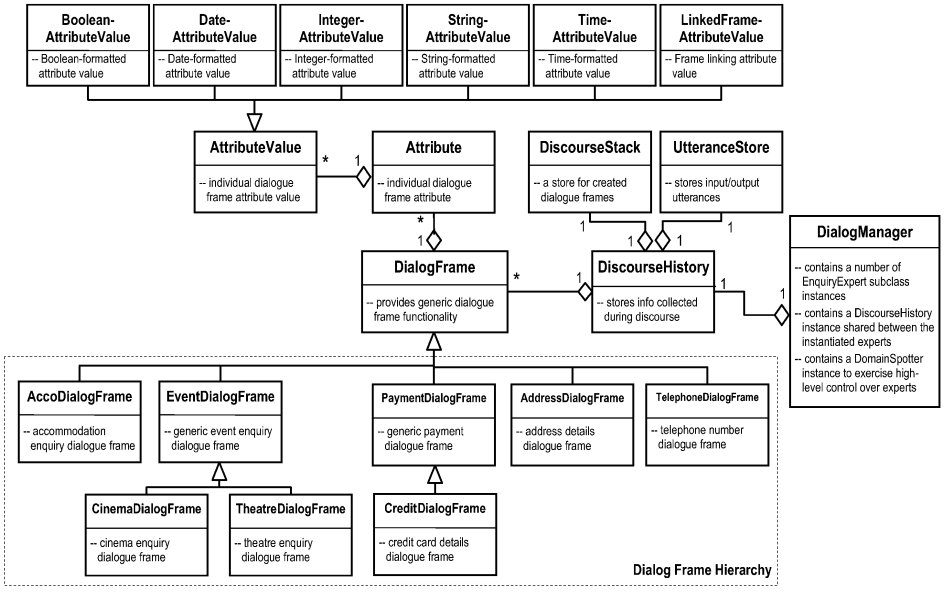

Fig. 2b. The Knowledge Hierarchy.

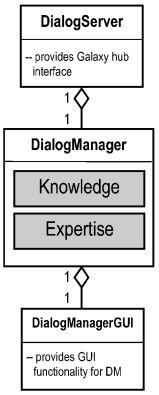

Fig. 2c. The DialogManager in context.

Rather than attempt to show all the classes in one class-relationship diagram, we have provided two views of the system, the DialogManager class serving as the element common to both views. Fig. 2a represents the Expertise Hierarchy, comprising components that the system uses to implement its generic confirmation strategies (i.e. what to do when the user supplies new, modified or negated information), and its domain-specific, dialogue-furthering strategies (i.e. what to ask for next—in an accommodation enquiry, or a theatre reservation, or when eliciting credit card details, for example). Fig. 2b on the other hand

represents the Knowledge Hierarchy. This hierarchy comprises components that store the data values that the system has elicited from the user, or has inferred for itself, in the course of a dialogue (for example the name of a hotel, the date when the user will arrive). More than this, the components in the Knowledge Hierarchy maintain information that indicates how confirmed each datum is, and what the system intends to do (if anything) to confirm that datum adequately. They are the system's discourse knowledge, and are used by implemented experts of the Expertise Hierarchy as they prepare system utterances in accordance with their generic and domain-specific rules.

Fig. 2c completes the picture by setting the DialogManager class in the context of the overall dialogue management module. One the one hand, in a development environment, the DialogManager needs to be able to report its view of the current discourse state and its dialogue-furthering decisions to the development team: thus we have given it a graphical user interface (GUI), and an associated text-based log file, which we use to monitor the system as it works. On the other hand, the DialogManager must be able to communicate with the components of the broader natural language system (the speech recogniser, the parser, and the other modules connected to the Galaxy hub). The DialogServer, as mentioned in Section 4, implements the interface to the Galaxy hub, through which the DialogManager sends information to and receives information from the other system modules. In the following subsections we provide descriptions of key features of the main system components.

5.1. DialogServer and DialogManager

DialogServer provides an interface to the hub. It contains a DialogManager, which as well as coordinating dialogue turn-taking, has a suite of business domain experts (AccommodationExpert is one example), grandchildren of DiscourseManager. DialogManager also has a DomainSpotter that helps select domain expertise, and a DiscourseHistory that maintains a record of user–system interaction across domains.

5.2. DiscourseManager

The DiscourseManager is responsible for the DM's overall 'conversational style', its generic discourse behaviour. Given the current difficulties of speech recognition, and the possibility that a user will misunderstand or change his or her mind, any system conducting a complex transaction must have a strategy for confirming the semantic contents of the user's utterances. The DiscourseManager determines the system's response to new, modified or negated information from the user, and it determines when domain-specific rules of thumb, encapsulated in the suite of domain experts, should be allowed to fire. Since the system's confirmation strategy is an implicit one, the system's utterances typically take the form of a confirmation of *new* or *modified* information supplied by the user (in a fully generated form the system utterance might be "So, you're arriving at the Hilton in Belfast on Friday—") followed immediately by the system's next question ("—what day are you departing?"). If the user has *negated* information that the system had previously recorded, the system's next utterance concentrates on correcting the negated information rather than seeking further information. Because the DiscourseManager is at the top of the inheritance hierarchy, its behaviour colours the manner in which its grandchildren

(the EnquiryExpert subclasses such as AccommodationExpert) interact with the user in their specialised business domains (accommodation booking, taking payment details, etc.).

The DiscourseManager is able to work out what has been repeated, modified or negated by the user by comparing a frame of information relating to the user's latest utterance, with a corresponding frame representing the last discourse state. This last discourse state indicates what information had been provided by the user, the status of each piece of information (repeated, modified, negated, etc.) and the system's previous intentions for confirming or repairing supplied or missing information. This last point is important: the system must be able to interpret the user's latest utterance in the light of its own last utterance (e.g. if the user answers "Yes" in response to the system's observation "So, you're staying in Dublin", then Dublin can be regarded as the confirmed location). Currently, if the system fails to identify the information it requests, it simply repeats its request if it has no new information to consider. In a business-strength implementation, repeated failure to acquire requested information might result in the system rephrasing its request, seeking alternative information, or ultimately abandoning the enquiry. We are currently exploring possibilities of linking the system's domain-independent dialogue behaviour with its speech recognition confidence—i.e. the more confident the system is that it knows what the user is saying, the freer the dialogue interaction; less confidence would mean a dialogue more closely led by the system, possibly to the level of requiring 'yes' or 'no' responses from the user to quite specific questions.

The DiscourseManager makes use of a number of other components. In order to let each of its domain-specific EnquiryExpert grandchildren update the record of the evolving dialogue, it takes on the DialogManager's DiscourseHistory as an inheritable attribute of its own. DiscourseHistory maintains a record of the evolving dialogue in the form of a stack of DialogFrames, each of which in turn comprises a set of Attribute objects relevant to the particular business domain. EnquiryExpert and its subclasses represent the domain-specific expertise that augment the behaviour of DiscourseManager once the latter's generic confirmation strategies have been applied.

5.3. DiscourseHistory and DialogFrame

Maintaining a record of the evolving discourse, and providing the means of creating and retrieving entries for individual user–system exchanges, are the responsibilities of the DiscourseHistory. The DiscourseHistory incorporates a stack of DialogFrames, and in the Java implementation includes methods (*addFrame*, *getLastFrame*, *getLastMatchingFrame*, etc.) that assist the DiscourseManager in the tasks of adding frames to and retrieving frames from the stack.

The DialogFrame is a set of attributes (of class Attribute—more about this presently) that corresponds to the frame of slots that must typically be filled to complete a transaction in a particular domain—accommodation booking or payments, say. The generic DialogFrame has methods that are not domain specific and that enable calling objects to (among other things) *addAttribute* and *getAttribute*.

Let us consider a specialisation of DialogFrame. AccoDialogFrame (a DialogFrame for handling accommodation enquiries), has a constructor that tags the frame with the *identifier* "Accommodation" (using generic DialogFrame's *setIdentifier* method), and uses

Table 1

DIALOGUE MANAGER: ATTRIBUTE OBJECT STRUCTURE	
'Att-atts' (Attributes of an Attribute Object)	**Typical value / usage**
String attributeName	e.g. "AccommodationName"
Object attributeValue	Potentially a String such as "Hilton Belfast".
int confirmationStatus	NEW_FOR_SYSTEM = 1; ... REPEATED_BY_USER = 3; ..., etc.
int discoursePeg	an integer that is incremented or decremented as a user confirms, modifies or negates a value—used in association with *confirmationStatus* above as an indicator of 'confirmedness'.
int systemIntention	how the system will respond given the confirmedness of a particular attribute—CONFIRM = 1; ... SPECIFY = 4;... etc.

DialogFrame's *addAttribute* method to initialise the frame with a number of attributes, such as might be used for an accommodation booking: accommodation type, date from, date to, etc. By tagging all instances of AccoDialogFrame with the *identifier* "Accommodation", we give ourselves the option of using DiscourseHistory's method *getLastMatchingFrame* to retrieve a frame that furthers a particular discourse strand (an accommodation enquiry, say), even though the user may have made other intervening utterances (about hiring a car, for instance) that cause frames pertinent to a different type of enquiry, albeit a related one, to be 'interleaved' among the AccoDialogFrames in the DiscourseHistory's stack.

5.4. Attribute

The objects of class Attribute, which are contained in an array of Attributes within a DialogFrame, are of particular interest. The agents, whether they provide service or support, collect and manipulate frames of information related to their own sphere of competence. The frames consist of Attribute objects, each of which stores:

- the name (e.g. accommodation name, accommodation class) and elicited value (whether a string, integer, etc.) of a single piece of information (datum);
- the confirmation status of the datum (e.g. new_for_system);
- the level to which the datum has been confirmed (through repetition, or by the user's affirmative response to a system prompt—the level is represented by a simple numeric 'peg');
- and the system intention regarding the datum (e.g. implicitly confirm new information; explicitly confirm information that has been negated; ask the user to specify information that is still required) [4].

The Attribute objects thus give a multi-faceted view of each piece of information that is being considered by the system: what its value is, how confirmed it is, what the system intends to do about it next. Table 1 gives an overview of the typical Attribute object structure in the Java implementation. Each object of type Attribute has in effect

its own set of Java attributes: we sometimes refer to these 'attributes of Attribute objects' as 'att-atts' for short.

The att-atts are used by the DiscourseManager's generic confirmation strategy, which involves creating new DialogFrames, based on a comparison of the semantic contents of the latest user utterance, with the contents of the last matching Frame taken from DiscourseHistory's stack. Thus the agent is able to determine which values have been confirmed (e.g. the user has not challenged an implicit confirmation request by the system) and which have been modified or negated. The frames of information are typically populated in the course of several discourse turns, as new or additional information is acquired from successive user–system interactions.

For example if an Attribute with the attributeName "AccommodationName" had the s*ystemIntention* CONFIRM in the last DialogFrame, and the user repeats the *attributeValue* ("Hilton Belfast", say), then the DialogManager's rules for evolving a new DialogFrame will state that, in the corresponding Attribute of the new DialogFrame, *discoursePeg* should be incremented, *confirmationStatus* should be set to REPEATED_BY_USER and that the system should reset its *systemIntention* for this Attribute to UNDEFINED (i.e. 'no further intention to confirm at this stage').

5.5. EnquiryExpert and EnquiryExpert subclasses

For each business area within the system there must be functionality (a) to decide what information to elicit next, or what information to infer, given that certain information may already have been provided, (b) to check the validity of the combinations of information provided, (c) to give the most helpful guidance when the user is having difficulty completing the enquiry, and (d) to decide when sufficient, confirmed information has been provided to conclude the transaction. While the sequencing of these operations can be encoded in a generic EnquiryExpert, detailed rules of thumb are specific to the EnquiryExpert subclasses (e.g. AccommodationExpert), which recreate in sometimes quite extensive sets of domain-specific heuristics the kind of behaviour that would characterise any human expert in a particular business domain—a cinema booking clerk or an accommodation adviser, for instance (e.g. 'if a cinema and a day have been given, ask the user to choose from a list of shows'; 'if a hotel has been confirmed but no dates have been given, ask for dates'). The domain specialists or 'Experts' within our system—AccommodationExpert, TheatreExpert, CinemaExpert, CreditCardExpert, etc.—all inherit generic dialogue-handling skills from the DiscourseManager. The domain experts themselves encapsulate specialised behaviour, which can be readily extended by additional classes. There are two families of domain experts:

- 'service agents' that provide front-line services to the user—such as AccommodationExpert, and
- 'support agents' such as CreditCardExpert that are able to elicit information required to complete one of the front-line service transactions.

We refer to the corresponding discourse segments as 'service' and 'support' dialogues respectively. By assigning the agents (and the corresponding dialogues) to one of two families we give ourselves the option of restricting user-led transitions between main and

ancillary transactions. However, the overall objective of our implementation is to maintain a high degree of flexibility in the manner in which the system reacts to unsolicited user utterances.

In addition to its inherited confirmation strategies, each of the domain Experts, whether a service agent or a support agent, has its own expert rules, contained in one or more expert rule sequences. Typically the expert rule sequences will be of one of three kinds:

- *user-focussed rules*: rules that are used to trigger the system's immediate response to specific confirmed combinations of information supplied by the user and recorded in the evolving dialogue frame—the rules may cause the system to ask for more information, or may initiate a database search.

  e.g. IF (the user has not given
  accommodation name [e.g. 'Hilton', 'Holiday Inn', etc.]
  and *accommodation type* [e.g. 'hotel', 'guesthouse', etc.])
  THEN ask for *accommodation type* (3)

- *database-focussed rules*: rules that are applied in the light of the system's failed attempts to retrieve information from or validate information against the database. These failed searches may result from a particular combination of search constraints, whether these are supplied by the user, or by the system when it attempts to retrieve information to assist the user. The database-focussed rules may therefore recommend that a constraint (e.g. the class of hotel) be relaxed in order to get a database match for other user requirements (e.g. the hotel location that the user has requested). The database-focussed rules represent recovery strategies that enable the system to offer viable alternatives when an enquiry might otherwise reach an impasse. The user remains free to reformulate the enquiry in a way that differs from the system's suggestion; indeed, in circumstances where the system has no specific recommendation to make, the system will simply explain why the database search has failed and pass the initiative back to the user.

  e.g. IF (failed search was to find accommodation name
  [e.g. Hilton, Holiday Inn, etc.]
  AND constraints were *location Belfast* and *class four-star* and
  accommodation type hotel)
  THEN relax constraint *class four-star* and re-do search (4)

- *housekeeping rules*: when the user changes key information in an enquiry (information needed to complete a transaction), the system resets—thus the notion of 'housekeeping'—the flags it uses to record the user's intention to proceed with the transaction, as well as the flag it uses to record its own intention to announce the conclusion of the transaction. At critical junctures in the discourse, the effect is to allow the system's normal confirmation strategies to be reapplied (e.g. if the user has given all key values required to complete the transaction, explicitly confirm those values), and to allow the user to reconfirm that he or she wishes to proceed with the transaction (e.g. in an accommodation enquiry, does the user still want the system to check availability,

or to attempt to make a reservation?). Housekeeping rules can be written for all key values in a transaction, and in any of its dialogue turns the system will allow as many rules to fire (i.e. perform as much housekeeping as is necessary) to take account of the current discourse state. Housekeeping rules prevent the conclusion of discourses on the basis of inadequately confirmed user intentions.

e.g. IF the user has changed *accommodation name*
[i.e. accommodation name has been *specified* but its confirmation level, as represented by its discourse peg, is now set to 0]
THEN reset 'check availability' flag, 'reserve' flag, and 'conclude transaction' flag (5)

The user- and database-focussed rules and housekeeping rules that are encapsulated in the domain experts are representative of the kinds of decision making that characterise a human expert in the particular domain—a booking clerk at a theatre, or a desk clerk at an hotel. The development team is continuing to add to and refine the rules in the light of observed human–human dialogue behaviour. For example, in (4) above, it might on occasion be preferable to search for a different hotel *location*, while maintaining the *class* constraint.

Within each domain expert, each rule is specified declaratively. For example, (3) above appears as

```
String userFocussedRule = "
[RULE]
 { AccoName UNSPECIFIED }
 { AccoType UNSPECIFIED }
[ACTION]
 { INTENTION AccoType SPECIFY }
[RULE-END]";
```
(6)

while (4) above appears as

```
String dbFocussedRule = "
[RULE]
{ AccoName TARGET }
 { AccoType CONSTRAINING }
 { Location CONSTRAINING }
 { AccoClass CONSTRAINING }
[ACTION]
 { RELAX \" AccoClass \" }
[RULE-END]";
```
(7)

and (5), preconditioned on a specified but as yet unconfirmed critical value (i.e. the value has been given by the user, but as yet its discourse peg is 0), is coded as

```
String housekeepingRule = "
"[RULE]
 { AccoName SPECIFIED }
 { AccoName PEG_EQ \" 0 \" }
[ACTION]
 { INTENTION AccoName CONFIRM }
 { REINITIALISE CheckAvailability }
 { REINITIALISE Reservation }
 { REINITIALISE Completion }
[RULE-END]";                                                    (8)
```

Specifying rules declaratively in this manner recreates some of the intuitiveness of rule-based programming—the suite of rules can be easily extended or reduced to capture the subtlety of human behaviour. In creating the rules the developer is not so much concerned with how the behaviour will be implemented as with what the behaviour should be.

However, implementing the behaviour needs to be addressed somewhere. The rule specifications are used as parameters for building ExpertRule objects, which contain methods for extracting and analysing the contents of the rule, and these rule objects are in turn built into ExpertRuleSequence objects (typically, for each domain expert, there will be a sequence for user-focussed rules and another for database-focussed rules). Each instance of EnquiryExpert (whether an AccommodationExpert, an EventExpert or a still more specialised subclass) is permitted by the generic confirmation strategy to test its rule sequences when there are no user-initiated negations to be addressed, with more specialised expert rule sequences tested before any more general, inherited, expert rule sequences. A user-focussed rule may thus cause a SPECIFY intention to be set against an attribute in a dialogue frame, or it may initiate a database search, and if this search fails to return the value(s) sought, the query may be resubmitted in amended form in accordance with the expert's database-focussed rules. System output is currently in the form of key phrases—so an implicit confirmation followed by a SPECIFY intention might be output as: "MESSAGE: GIVEN [AccoType] Hotel; Specify AccoLocation." The natural language generation module that is currently being integrated into the system will accept this or similar semantic output and generate a well-formed utterance.

5.6. DomainSpotter: finding the right agent

5.6.1. Appointing an initial handling agent

To begin the dialogue, in order to identify the most appropriate 'handling agent', the DomainSpotter supplies each *service* agent with the output of the semantic parse. As it attempts to find an initial handling agent, the DomainSpotter considers only service agents (such as AccommodationExpert or CinemaExpert) and not support agents (such as CreditCardExpert or AddressExpert). The service agents represent the primary transaction types (booking a hotel room, enquiring about a movie, etc.) that the system handles: the system is not, for example, intended to allow the user to process their credit account, though it may elicit credit card details *in support of* a service (a hotel booking for instance). Such restrictions help the system ground its primary functions with the user. Each service

agent scores the parse of the initial user utterance against the semantic categories that it can process (each agent has a range of integer values—degrees of relevance—that it will assign to different domain-specific parse tags) and returns the score to the DomainSpotter. The service agent that scores highest is the one that the DialogManager asks to apply its domain-specific heuristics to the more detailed processing of the enquiry. For example, an AccommodationExpert might score highest and so become the handling agent if the user has been asking about hotels in Belfast. Specialised agents give a higher score for specialised parser tags than generic agents. For example, a user request "I'd like to go to see THE LION KING 2." might parse as: *event_enquiry:[Event_type].[Movies].THE LION KING 2*. Although the EventExpert could award a score for *event_enquiry*, the CinemaExpert, as a child of EventExpert, would award a score not only for *event_enquiry*, but for *Movies* as well, and so would be the winner.

5.6.2. Identifying the system's expertise

If the DomainSpotter is unable to identify a winning agent, it will ask the user to choose between the domains in closest contention. Indeed, if the user's enquiry is so vague as to give no domain-related information ("I'd like to make an enquiry."), the DomainSpotter will ask the user to choose between one of its top-level service agents: e.g. "Please choose between *event booking* or *accommodation booking*."—where the words in italics are actually provided by the service agents. The DomainSpotter is in effect relaying to the user information that the system components know about themselves: it is part of the system's design philosophy that higher level components are largely ignorant of the precise capabilities of lower level components. Similarly, if a service agent needs to avail itself of a support agent in a particular area, it tells the DomainSpotter to find it an expert that handles the particular specialism (*payments*, for instance): it does not name a specific expert object. So that its area of expertise can be identified, each agent has, as one of its attributes, a vector of the specialisms it deals with. The intention is that additional lower level expertise can be added to the system in such a way that higher level behaviour (i.e. requesting the expertise) remains unchanged. Where more than one expert (e.g. CreditCardExpert and InvoiceExpert) can deal with the requested specialism (e.g. *payments*), the DomainSpotter asks the user to choose.

5.6.3. Moving between service and support dialogues

In order to maintain the enquiry focus we use an ExpertFocusStack in the DomainSpotter. Once an agent is selected to handle the current discourse segment, it is pushed on to the top of the stack. The agent then uses its expert rules to elicit all the information needed to complete its discourse segment: an AccommodationExpert, for example, will be looking for all information needed to complete an accommodation booking. Depending on the rules it encapsulates, a *service* agent may require help from a *support* agent. For example, if an AccommodationExpert has confirmed sufficient information to proceed with a reservation, it will request help from an agent whose specialism is *payment*, and the DomainSpotter will look for one.

Let us pursue this example further. The PaymentExpert is identified as an appropriate payment handler, and is placed above AccommodationExpert on the ExpertFocusStack. However, let us suppose that eliciting payment details first involves eliciting address

details, and so the PaymentExpert in its turn asks the DomainSpotter to find it an agent specialising in address processing—in this case the AddressExpert. The AddressExpert now goes to the top of the ExpertFocusStack, above the PaymentExpert. Just like any other agent the AddressExpert has its own rules that allow it to accept typical combinations of information supplied (prompted or unprompted) by the user and to ask appropriate follow-up questions for whatever information is still missing. Once a support agent has all the information it needs, one of its rules will fire to 'pass control back', along with a 'finished' message, to whatever agent was below it on the ExpertFocusStack. Thus AddressExpert will pass control back to PaymentExpert in this example, whose rules, if the user does not introduce a new topic, will continue to fire until all necessary payment information has been elicited and the payment subdialogue can be concluded—at which point control is passed back to the AccommodationExpert.

5.6.4. Dealing with user-led focus shifts

A mixed initiative dialogue manager needs to be able to cope with user-initiated shifts of discourse focus. For example, a user may supply credit card information unprompted, while the system's intention is first to elicit address information. At present we permit transfer of dialogue control between service agents: a user may, for example, want to discuss an event booking more or less in parallel with making accommodation arrangements. However, in order to ground the dialogue by eliciting information in a definite context, we impose some restrictions on user-initiated shifts of focus between support dialogues, and between support and service dialogues. For example, analogous to human–human dialogues, it may be desirable to hold the dialogue focus on a support dialogue, such as gathering payment details for a confirmed accommodation booking, rather than interrupt the support dialogue to start a new service enquiry, about cinema bookings, for instance. Dialogue frames are instrumental in implementing such policies.

Dialogue frames help identify the support dialogues associated with each service dialogue: the specification of each frame type (e.g. an AccommodationDialogueFrame) indicates the type of each of its Attributes, some of which may themselves be links to other frames (e.g. a PaymentDialogueFrame). Dialogue frames that are associated with service dialogues can be expanded into a tree-like structure by recursively traversing the various support frames that are linked to the service dialogue frame. For those frames which have already been in the discourse focus (i.e. frames representing dialogue tasks that have already been the subject of user–system interaction), this is a straightforward task. Additionally the frames of possible future handling agents can be predicted and included within the tree through the use of the DomainSpotter. For example, at the outset of an accommodation enquiry, the related service dialogue frame will not generally contain an explicitly linked payment frame. However, the DomainSpotter is able to determine which agents can provide payment support, and so the system generates a number of potential discourse paths relating to payment. Key words in the user's utterance determine which path is in fact used and which payment-related frames are explicitly linked to the accommodation frame.

As the dialogue evolves, the DomainSpotter tests which agents are best placed to handle the user's last utterance. The last utterance may of course relate to the domain dealt with by the current handling agent (e.g. the system may be dealing with accommodation details, and the user may, helpfully, supply information that also pertains to accommodation).

However, when the user's utterance contains information that falls outside the domain of the current handling agent ('out-of-domain' information), the tree of dialogue frames associated with the current service dialogue frame indicates to the DomainSpotter which support agents have been or may be involved in the current service enquiry, and should therefore be considered as handlers for the last utterance (e.g. the user may have provided a telephone number when the AccommodationExpert is still asking about the type of accommodation required, but a telephone number may still be relevant to the broader accommodation transaction—it may be needed when the user gives personal information regarding payment).

Let us consider the permitted and prohibited shifts of discourse focus between the agents within the system. Fig. 3 gives an overview of the DomainSpotter's decision-making mid-transaction. If the user's last utterance is scored most highly by a support agent which is relevant to the current service and whose topic has *already* been in the discourse focus, the user can return to this topic (the shift may indicate the user's intention to add to or modify information that was previously supplied). As a safeguard, the system will reorder the ExpertFocusStack in these circumstances, so that any support agents whose rules fired on the previous path to the revisited agent will be allowed to test their rules again (new address information, for instance, may affect a credit card option—e.g. if the revised address is in the UK, the CreditCardExpert may mention UK cardholder offers).

Other requests for shifts of focus from and to support agents are generally deferred, until the rules of the current handling expert request transfer ("Thanks, I'll take the telephone details in a moment…"). The system does not ignore the contents of the utterance that led to the deferral: the DiscourseHistory contains an UtteranceStore, a stack of the parses of the user's utterances. When it takes control of the dialogue (e.g. because one of the handling expert's rules has requested its services), an agent first looks to the UtteranceStore to see if there is any unprocessed information that it can handle. If there is, it takes the unprocessed parsed information and begins its processing as usual with its inherited confirmation strategy and its domain-specific expert rules ("You mentioned a telephone number. Let me just confirm the details: area code…").

If the DomainSpotter fails to locate a potential handling agent for an 'out-of-domain' utterance in the context of the current service transaction, it will poll the other service agents (does the user want to change from an accommodation enquiry to asking about cinema bookings?). Before transferring to (or indeed, before refusing a transfer to) a new handling agent—whether a service agent or a support agent—the DomainSpotter will always confirm the user's intentions ("Do you want to enquire about cinema bookings?"): the developers' intention is to avoid serious misinterpretations of users' free-form utterances, given that the system has the ability to work in a number of different though potentially related transaction domains. When deferring a request for a new service enquiry the DomainSpotter places the relevant expert on the bottom of the ExpertFocusStack, so that it will come into the discourse focus later.

5.7. Using Java objects to pass information across the Galaxy hub

A further element of the object-oriented solution is the means by which the DialogManager communicates with the database server via the Galaxy hub. Whenever an

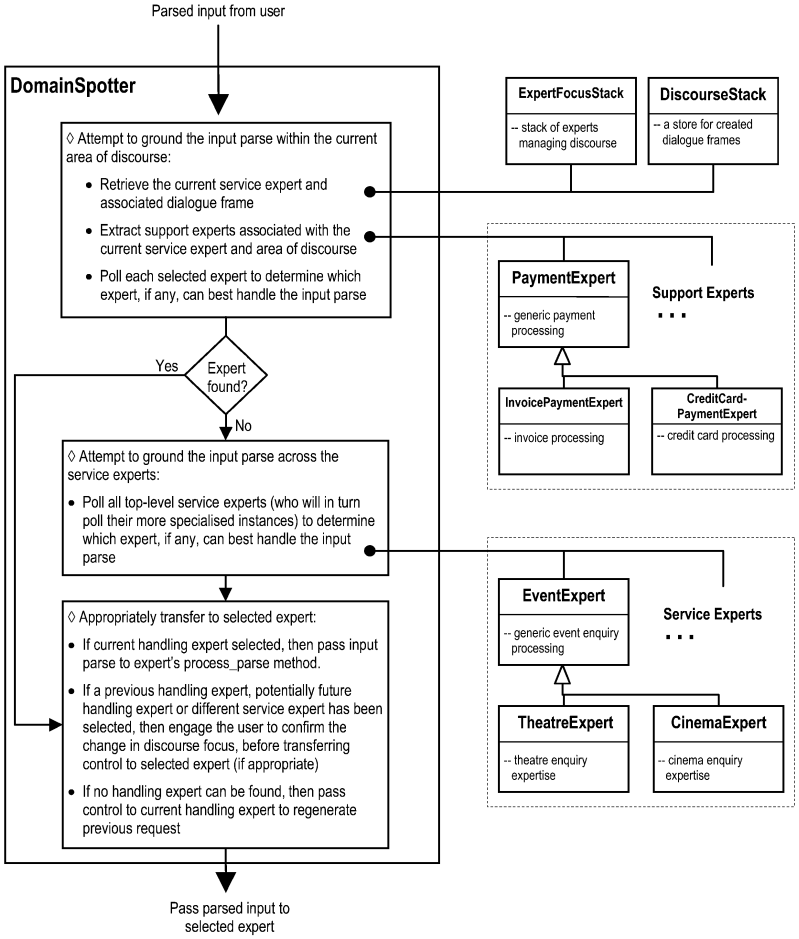

Fig. 3. Domain spotting to further an on-going transaction.

EnquiryExpert subclass needs to make a database search, it creates a DBRequest object whose attributes record which values are sought, which search constraints are to be used for the database search, and which constraints have been relaxed (i.e. require new values). The object must pass between two servers (going from the DialogServer to the DatabaseServer and back again) via the Galaxy hub. The DBRequest class therefore includes the encoding and decoding functionality that allows its instances to be encapsulated at the DialogServer as a bitstream within a Galaxy hubframe and reconstituted at a receiving DatabaseServer as an object. The contents of the DBRequest object are then used to formulate an SQL database query. The DBRequest object is populated with the results of the database search. It is encoded again and sent back via the Galaxy hub to the dialogue manager, where it is reconstituted and passed back to the domain expert that initiated the search. The domain expert can then apply its rules of thumb to the data in the DBRequest object. Fig. 4 gives

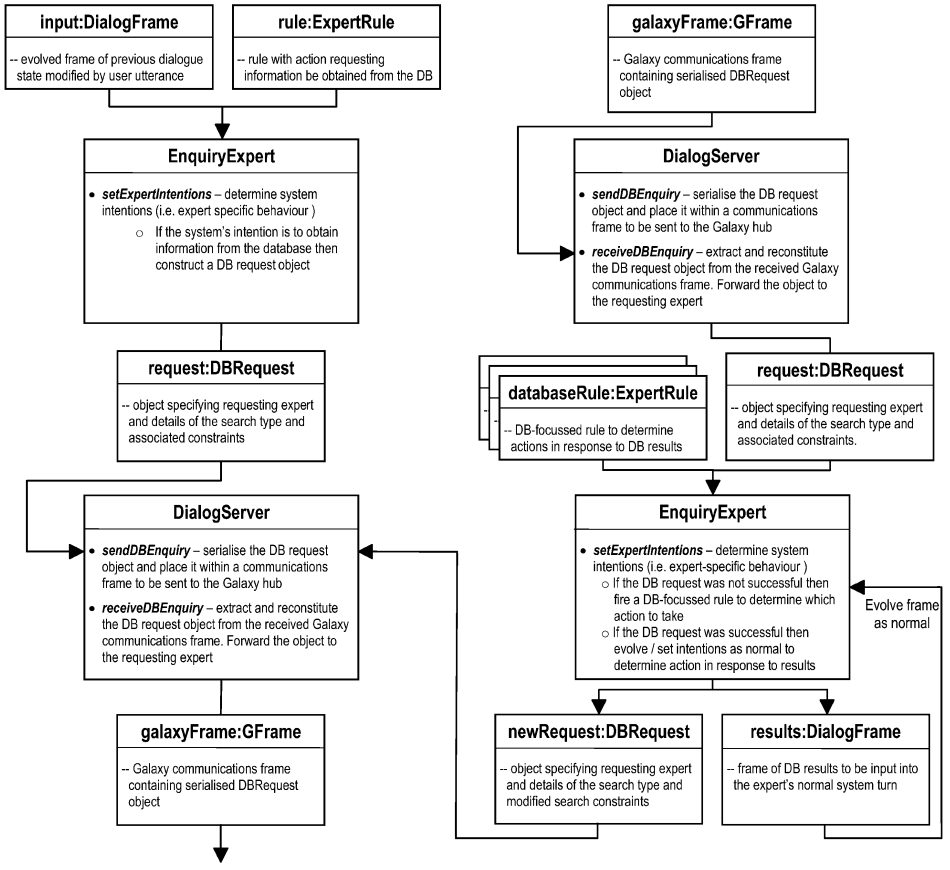

Fig. 4. Sending information across the Galaxy hub.

a diagrammatic representation of the process. Of note is the fact that information retrieved from the database, if it indicates that a user's request cannot be satisfied, may cause a further DBRequest to be generated (the result of a database-focussed rule having fired, in order to reformulate a failed query and so provide the user with alternative values from which to choose—e.g. there may be no vacancies on the date on which the user wished to stay at a particular hotel, so the system relaxes the hotel constraint and queries the database for an alternative hotel).

6. The Java dialogue manager in action

The comment box in Fig. 5 describes the manner in which the system evolves the discourse by combining its generic and domain-specific intentions: a handling agent first checks what generic confirmation issues should be addressed (for example, are there new or changed user-supplied values to be confirmed?) and then checks if it can fire any of its domain-specific rules, first housekeeping rules and then user-focussed and

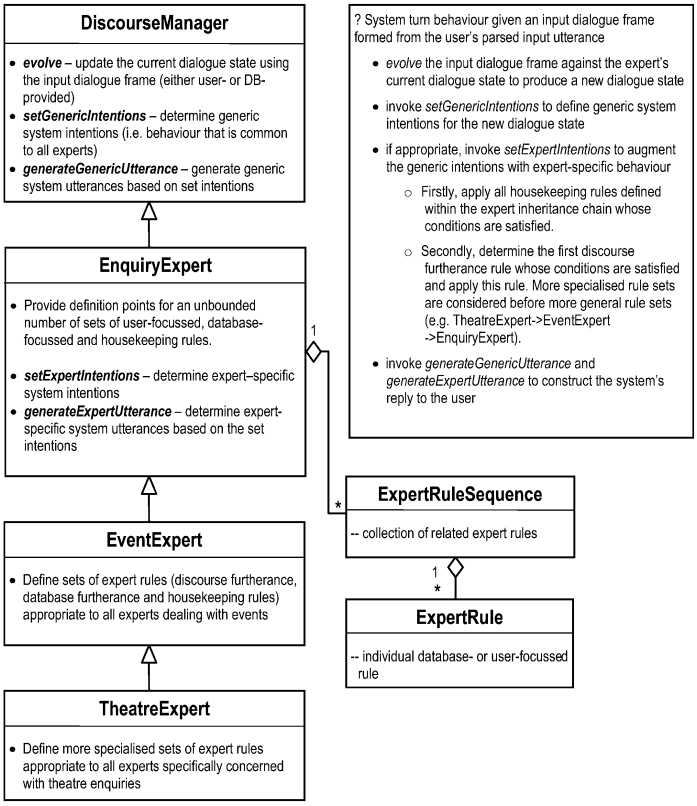

Fig. 5. Interplay of generic and domain-specific behaviour.

database-focussed rules—these last two flavours of rule constituting 'dialogue furtherance rules', in that they indicate to the user the information that the system needs to further the transaction, and, in the case of database-focussed rules, they generally suggest values from which the user may wish to choose (names of hotels with vacancies, for example).

The current prototype system accepts keyed natural language input and forwards its output constructs via an NLG (natural language generation) module—which for testing performed only this forwarding function—to the TTS (text-to-speech module). Thus system output is in the form of spoken semantic constructs rather than well-formed natural language utterances. Furthermore, the dialogues in each of the transaction domains covered by the system are simplified versions of the actual exchanges one might expect in a 'business-strength' application. For the moment, however, such dialogues serve to illustrate the manner in which discourse handling passes to the various service and support agents within the dialogue manager, and the manner in which those agents use their inherited and their own domain-specific dialogue management functionality.

The following scenario, and the corresponding conceptual DM inputs and outputs, gives a flavour of the system in action.

Scenario

A user wishes to book a four-star hotel in Belfast from 15–20 December. When the system returns a list of four-star hotels, the user changes the request to a three-star hotel. The system returns a list of three-star hotels and the user selects one and asks to book a double room. The user then asks for a single room rather than a double. As the system elicits payment details the user asks about movies. The system defers the movie enquiry, concludes the accommodation booking and then takes up the user's cinema enquiry, 'remembering' the cinema-related information that was supplied earlier.

A transcript of the user–system dialogue corresponding to this scenario now follows. While the user utterances may be input into the system as written, the well-formed system utterances are the developers' glosses on the actual system utterances, which take the form of the semantic constructs and which are also given. A brief commentary, indicating which main features of the Java DM are being called into play, is given at key points in the dialogue and is printed in italics.

S1: Welcome to the Queen's Communicator. How can I help you?

```
MESSAGE: StartUpAnnouncement.
```

U1: I'd like to book a four-star hotel in Belfast from the fifteenth of December to the twentieth of December.

Key words such as four-star and hotel fall within the accommodation domain, and are scored by the AccommodationExpert. Accordingly the DomainSpotter appoints the AccommodationExpert 'handling agent'. The AccommodationExpert populates its AccommodationDialogFrame with whatever relevant information has been supplied by the user.

S2a: So that's an accommodation booking—a four-star hotel in Belfast from the fifteenth of December two thousand and three to the twentieth of December two thousand and three. Please wait while I check the database...

```
MESSAGE: process accommodation booking
MESSAGE: GIVEN AccoType [ HOTEL ] AccoClass [ four-star ]
Location [ BELFAST ] DateFrom [ 2003-12-15 ] DateTo  [ 2003-12-20 ]
MESSAGE: database enquiry initiated
```

The AccommodationExpert, using the generic confirmation strategies that it has inherited from the DiscourseManager, implicitly confirms the new values that it has identified, and uses its own domain-specific rules to initiate a database lookup that will retrieve names of hotels that match the user's criteria.

S2b: You can choose between the following hotels: the Europa, the Malone Lodge, the Stormont or the Holiday Inn

```
PROMPT: SPECIFY AccoName [ europa ]  [ malone-lodge ]
[ stormont ] [ holiday-inn ]
```

U2: Could you make that a three-star?

Although the system in its utterance has strongly suggested that the user should select one of the hotel names it has retrieved from the database, this is a mixed initiative system, so the user may provide information different to that which was requested. In this case the user changes the star-rating required.

S3: OK, got you. That's three-star accommodation. Is that OK?

```
MESSAGE: user changed value AccoClass [ three-star ]
QUERY: confirm changes
```

Again, using its inherited confirmation strategy, the AccommodationExpert compares the value for the star-rating in the previous frame with the star-rating in the current frame and performs a 'repair-confirm' on the changed value.

U3: Yes, that's right.

S4a: Please wait while I check the database.

```
MESSAGE: database enquiry initiated
```

S4b: You can choose between the following hotels:

```
PROMPT: SPECIFY AccoName [ jurys ] [ lisdara ]
```

U4: I'll go for a double room at the Jury's.

S5: So, let me just check that. You want a double room at the three-star Jury's hotel in Belfast, from the fifteenth of December two thousand and three to the twentieth of December two thousand and three. Are those details correct?

```
MESSAGE: explicit confirm values AccoName [ jurys ] AccoType
[ HOTEL ] AccoClass [ three-star ] Location [ BELFAST ]
AccoRoomType   [ double ]   DateFrom   [ 2003-12-15 ]
DateTo [ 2003-12-20 ]
QUERY: CONFIRM values
```

In the current system implementation the handling agent will explicitly confirm all key values—i.e. those needed to conclude a major segment of the discourse—once these have been supplied by the user.

U5: Make that a single room.

Again however, the user changes his/her mind. Rather than perform a 'repair-confirm' on the individual changed value and re-confirm it in the general confirmation of key values, generic EnquiryExpert behaviour determines that the handling agent should perform only the general confirmation:

S6: So, let me just check that. You want a single room at the three-star Jury's hotel in Belfast, from the fifteenth of December two thousand and three to the twentieth of December two thousand and three. Are those details correct?

```
MESSAGE: explicit confirm values AccoName [ jurys   ] AccoType
[ HOTEL ] AccoClass  [ three-star ] Location [ BELFAST ]
AccoRoomType   [ single ]  DateFrom   [ 2003-12-15 ]
DateTo [ 2003-12-20 ]
QUERY: CONFIRM values
```

U6: Yes.

S7: Would you like me to check availability?

```
PROMPT: SPECIFY CheckAvailability
```

U7: Yes.

S8a: OK, I'll see if that's available.

```
MESSAGE: GIVEN CheckAvailability [ true ]
```

S8b: Yes that's available. Would you like me to reserve that for you?

```
MESSAGE: Availability OK
PROMPT: SPECIFY Reservation
```

U8: Yes, please.

S9a: OK, I'll see if I can reserve that.

```
MESSAGE: GIVEN Reservation [ true ]
```

S9b: Yes, that's reserved. Let's go on now with payment details. Let's start with the address. Can you tell me your house number please?

```
MESSAGE: Reservation OK
MESSAGE: process payment details
MESSAGE: process address details
PROMPT: SPECIFY HouseNumber
```

Currently we are using simple 'MESSAGE: process...' flags to indicate what area of discourse the system is addressing. These utterances are directly related to the underlying system implementation: the AccommodationExpert, having elicited the hotel details, now calls, via the DomainSpotter, on the services of a PaymentExpert, which in turn calls on the services of an AddressExpert. More sophisticated semantic constructs than these simple domain identifiers may be required as a basis for full natural language generation.

U9: It's 3 Malone Road, Belfast BT7 1NN.

Although the system asked only for the house number, the user provided his/her full address. In a frame-based, phrase-spotting system such as this, any relevant information will be used by the handling agent (at this point an AddressExpert) to populate slots in the current dialogue frame. The unrequested information is thus accepted...

S10: So, let me just check that. That's number three Malone Road; city: Belfast; postcode BT7 1NN. Are those details correct?

```
MESSAGE: explicit confirm values HouseNumber  [ 3 ] StreetName
[ MALONE ROAD ] CityName [ BELFAST ] PostCode  [ BT7 1NN ]
QUERY: CONFIRM values
```

U10: Yes.

S11: OK, I've got all your address details. Let's go on now with payment details. How would you like to pay?

```
MESSAGE: address details complete
MESSAGE: process payment details
PROMPT: SPECIFY PaymentType
```

U11: Could I just check what's on at the UGC Cinema?

Again the user takes the initiative by ignoring the request from the handling agent—which is now the PaymentExpert—for details of the type of payment (e.g. credit card or invoice). Instead the user asks about an entirely different transactional domain, namely cinema bookings. On the basis of key words in the user's utterance ('UGC', 'Cinema', which will result in the parse segment '[Event_venue].[Cinema].ugc', the DomainSpotter recognises that the user is attempting to change topic from a support agent to a new service agent, and at this stage defers the request. However, the DomainSpotter places the CinemaExpert at the bottom of the ExpertFocusStack for use later, and thanks to the system's UtteranceStore, the CinemaExpert, when it eventually becomes the handling agent, will be able to make use of the information that the user has already supplied.

S12: Do you want to enquire about cinema bookings?

```
PROMPT: SPECIFY UserUnderstood QUERY change domain [ cinema
booking ]
```

U12: Yes.

S13: Got it. I'll just complete the payment details first. We'll look at cinema bookings in a moment. Let's go on now with payment details. How would you like to pay?

```
MESSAGE: GIVEN UserUnderstood [ true ]
MESSAGE: transfer not possible from [ payment details ] defer
transfer to [ cinema booking ]
MESSAGE: process payment details
PROMPT: SPECIFY PaymentType
...
```

And now, once the payment details for the accommodation have been elicited and confirmed, and the accommodation booked, the system is ready to move on from the AccommodationExpert to the next agent on the ExpertFocusStack, in this case the CinemaExpert—which, like all agents, first of all checks the UtteranceStore for any relevant, previously parsed input.

S20: So that's the accommodation booking complete. Let's go on now with the cinema booking. You mentioned the UGC cinema. What date would you like to go?

```
MESSAGE: accommodation booking complete
MESSAGE: process cinema booking
MESSAGE: recall previous utterances
MESSAGE: GIVEN EventVenue [ ugc ]
PROMPT: SPECIFY Date
```

U20: The seventeenth of December.

S21a: So that's the seventeenth of December two thousand and three. Please wait while I check the database.

```
MESSAGE: GIVEN Date [ 2003-12-17 ] MESSAGE: database enquiry
initiated
```

S21b: Which of the following movies would you like to see: Brother Bear, Love Actually, Elf, SWAT, The Fighting Temptations, The Mother, Thirteen, Timeline, Lord of the Rings 3.

```
MESSAGE: GIVEN Date [ 2003-12-17 ]
PROMPT: SPECIFY EventName [ Brother Bear ] [ Love Actually ]
[ Elf ]  [ SWAT ] [ The Fighting Temptations ] [ The Mother ]
[ Thirteen ]  [ Timeline ] [ Lord of the Rings 3 ]
...
```

7. Looking ahead

We believe that an object-oriented approach to dialogue management using Java has considerable promise. We have already decomposed the cross-domain dialogue management task intuitively into a number of subdialogues, each conducted by an implemented domain specialist with its own expert rules and associated frame of information to collect. By using inheritance we easily establish a common approach to dialogue management, independent of domain: all experts inherit the same confirmation strategy. Through inheritance we ensure that domain experts have common characteristics: they all have sequences of 'expert rules' that they can apply to user-supplied information to determine what the system should do next. Domain spotting enables us to identify appropriate dialogue-handling expertise for each of the user's utterances. Since our DomainSpotter actively looks for relevant expertise amongst the cohort of service and support agents, new expertise can readily be added without disturbing the system's fundamental dialogue management strategies. Additionally, division of the available experts into (front-line) service agents and (ancillary) support agents helps us maintain discourse context by deferring user-led shifts of focus that interrupt coherent data elicitation. Java gives us the required OO structures to implement our suite of inheriting collaborating objects. Moreover, using the Java API to the Galaxy hub, we have ready access to off-the-shelf, hub-compliant components that are needed to support implementation of a complete spoken dialogue system.

Future development plans include the addition of new service and support agents to cover a broader range of mixed initiative dialogues (e.g. travel-related enquiries).

Accordingly our parser grammars will be extended to allow identification of more key words and phrases. Inclusion of an appropriate speech recognition engine and a natural language generation module will result in a system that can process spoken input and produce naturalistic spoken output, across our chosen business domains. From a software engineering perspective, a further avenue of enquiry will be an analysis of the OO design patterns that have emerged from our initial Java implementation, and an exploration of opportunities to implement established design patterns.

Acknowledgements

This research is supported by the EPSRC under grant number GR/R91632/01.

References

[1] J. Allen, D. Byron, M. Dzikovska, G. Ferguson, L. Galescu, A. Stent, An architecture for a generic dialogue shell, Natural Language Engineering 6 (3–4) (2000) 1–16.
[2] G. Booch, Object-Oriented Analysis and Design with Applications, 2nd edition, Benjamin/Cummings, Redwood City, CA, 1994.
[3] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison Wesley Longman, Reading, MA, 1998.
[4] P. Heisterkamp, S. McGlashan, Units of dialogue management: an example, in: Proceedings of ICSLP96, Philadelphia, 1996, pp. 200–203.
[5] http://communicator.colorado.edu.
[6] http://communicator.sourceforge.net/.
[7] http://fife.speech.cs.cmu.edu/Communicator/index.html.
[8] http://www.cstr.ed.ac.uk/projects/festival/.
[9] http://www.sls.csail.mit.edu/sls/technologies/galaxy.shtml.
[10] I.M. O'Neill, M.F. McTear, Object-oriented modelling of spoken language dialogue systems, Natural Language Engineering 6 (3–4) (2000) 341–362.
[11] I.M. O'Neill, M.F. McTear, A pragmatic confirmation mechanism for an object-based spoken dialogue manager, in: Proceedings of ICSLP-2002, vol. 3, Denver, September, 2002, pp. 2045–2048.
[12] I. O'Neill, P. Hanna, X. Liu, M. McTear, The Queen's Communicator: an object-oriented dialogue manager, in: Proceedings of Eurospeech 2003, Geneva, 2003, pp. 593–596.
[13] A. Rudnicky, W. Xu, An agenda-based dialog management architecture for spoken language systems, in: Proceedings of IEEE Automatic Speech Recognition and Understanding Workshop, 1999, p. I–337.
[14] M. Turunen, J. Hakulinen, Agent-based adaptive interaction and dialogue management architecture for speech applications, in: Text Speech and Dialogue—Proceedings of the Fourth International Conference TSD, 2001, pp. 357–364.
[15] W. Ward, Extracting information in spontaneous speech, in: Proceedings of ICSLP94, Yokohama, 1994, pp. 83–86.